Topics:

- Optimization
- Convolutional Layers

CS 4644-DL / 7643-A
ZSOLT KIRA

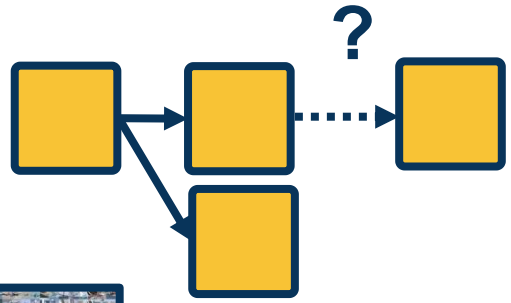
- **Quiz on Wed!!!**
  - Practice quiz and logistics out
  - Closed book/notes, no scratch sheets, no calculator.

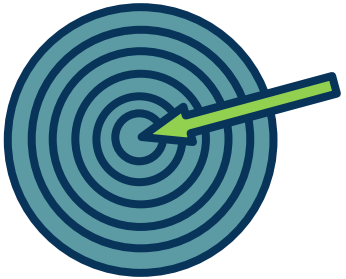
- **Assignment 2 due 02/22**
  - Implement convolutional neural networks

- **Project Proposal:** Out and due **Feb 14th**

There are still many design decisions that must be made:

- **Architecture**
- **Data Considerations**
- **Training and Optimization**
- **Machine Learning Considerations**





Local Minima

Solution: Time-varying bias correction

Typically $\boldsymbol{\beta_1 = 0.9,\ \ \beta_2 = 0.999}$

So $\hat{v}_i$ will be small number divided by (1-0.9=0.1) resulting in more reasonable values (and $\hat{G}_i$ larger)

$$v_i = \beta_1 v_{i-1} + (1-\beta_1)\left(\frac{\partial L}{\partial w_{i-1}}\right)$$

$$G_i = \beta_2 G_{i-1} + (1-\beta_2)\left(\frac{\partial L}{\partial w_{i-1}}\right)^2$$

$$\hat{v}_i = \frac{v_i}{1-\beta_1^t} \qquad \hat{G}_i = \frac{G_i}{1-\beta_2^t}$$

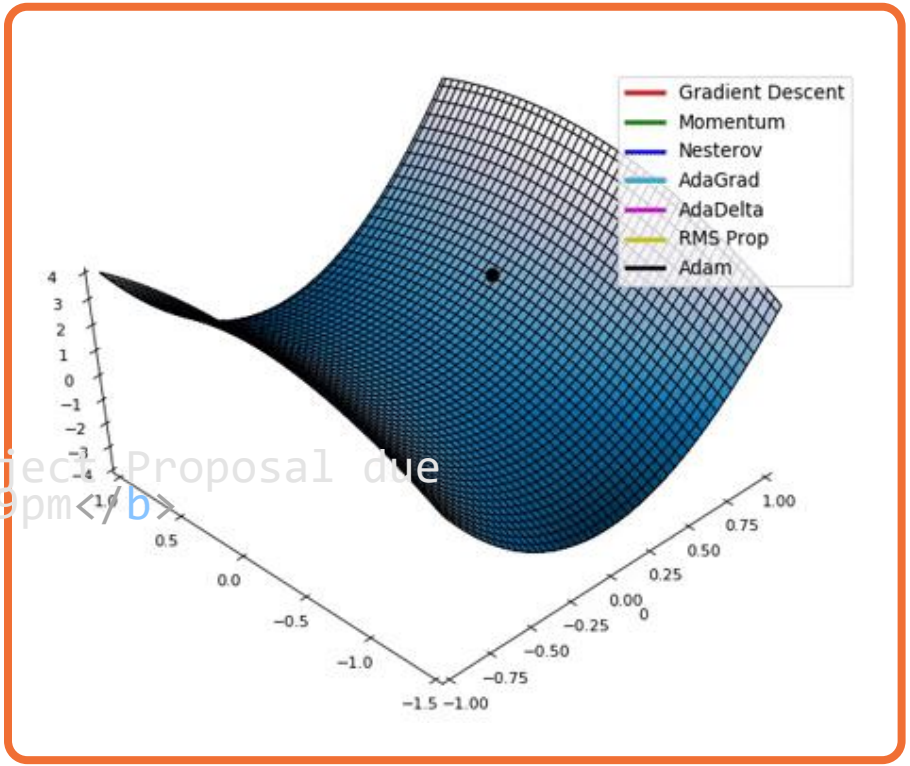
$$w_i = w_{i-1} - \frac{\alpha\, \hat{v}_i}{\sqrt{\hat{G}_i + \epsilon}}$$

Optimizers behave differently **depending on landscape**

Different behaviors such as **overshooting, stagnating, etc.**

Plain SGD+Momentum can generalize better than adaptive methods, but requires more tuning

- **See:** *Luo et al., Adaptive Gradient Methods with Dynamic Bound of Learning Rate, ICLR 2019*



From: https://mlfromscratch.com/optimizers-explained/#/

Initialization

Initializing the Parameters

The parameters of our model must be **initialized to something**

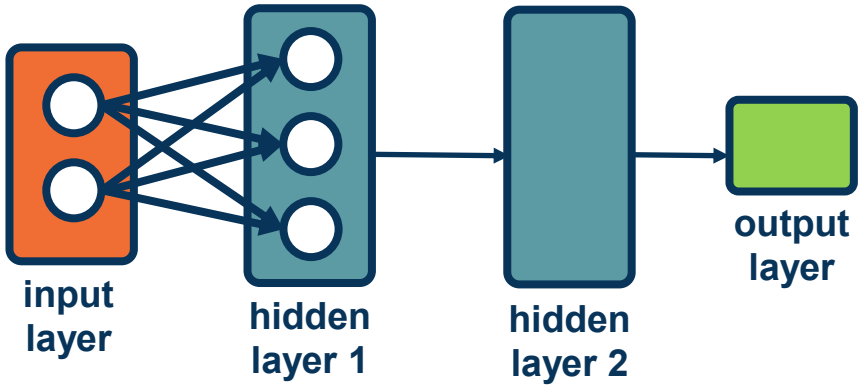
- Initialization is **extremely important**!
  - Determines how **statistics of outputs** (given inputs) behave
  - Determines how well **gradients flow** in the beginning of training (important)
  - Could **limit use of full capacity** of the model if done improperly
- Initialization that is **close to a good (local) minima** will converge faster and to a better solution

Georgia Tech

Initializing values to a constant value leads to **a degenerate solution**!

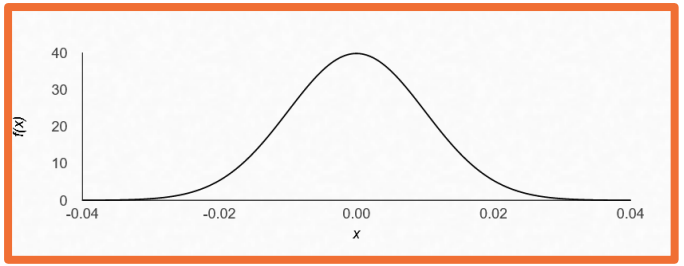
- What happens to the **weight updates?**
- Each node has the same input from previous layers so gradients **will be the same**
- As a results**, all weights will be updated** to the same exact values

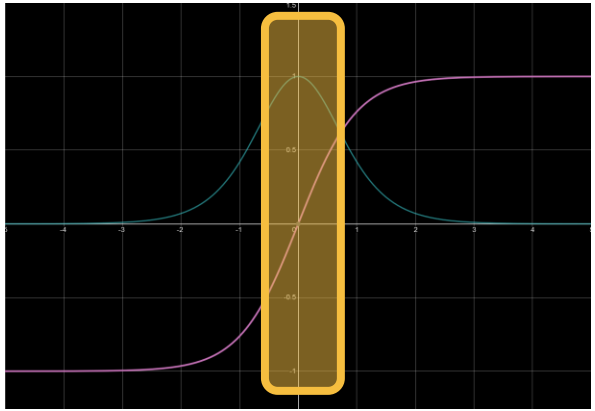
$$w_i = c \quad \forall i$$



Common approach is **small normally distributed random numbers**

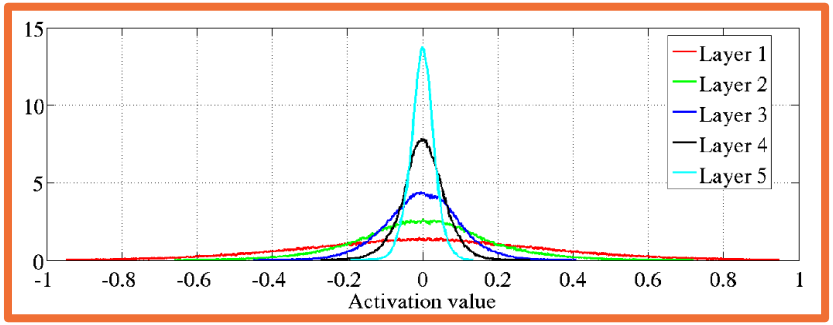
- E.g. $N(\mu, \sigma)\ where\ \mu = 0, \sigma = 0.01$
- **Small weights** are preferred since no feature/input has prior importance
- Keeps the model within the **linear region of most activation functions**





Deeper networks (with many layers) are more sensitive to initialization

- With a deep network, **activations (outputs of nodes) get smaller**
  - Standard deviation reduces significantly
- **Leads to small updates –** smaller values multiplied by upstream gradients



Distribution of activation values of a network with tanh non-linearities, for increasingly deep layers

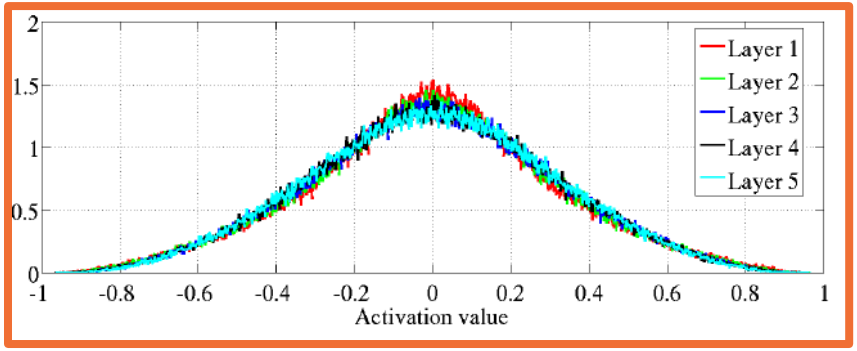
From "Understanding the difficulty of training deep feedforward neural networks." AISTATS, 2010.

Ideally, we'd like to maintain the variance at the output to be similar to that of input!

- This condition leads to a **simple initialization rule**, sampling from uniform distribution:

$$\text{Uniform}\left(-\frac{\sqrt{6}}{n_j+n_{j+1}}, +\frac{\sqrt{6}}{n_j+n_{j+1}}\right)$$

  - Where n_j is **fan-in** (number of input nodes) and n_{j+1} is **fan-out** (number of output nodes)



Distribution of activation values of a network with tanh non-linearities, for increasingly deep layers

From "Understanding the difficulty of training deep feedforward neural networks." AISTATS, **2010.**

In practice, **simpler versions** perform empirically well:

$$N(0,1) * \sqrt{\frac{1}{n_j}}$$

- This analysis holds for **tanh or similar activations**.
- Similar analysis for **ReLU activations** leads to:

$$N(0,1) * \sqrt{\frac{1}{n_j/2}}$$

"Delving Deep into Rectifiers:Surpassing Human-Level Performance on ImageNet Classification", ICCV, 2015.

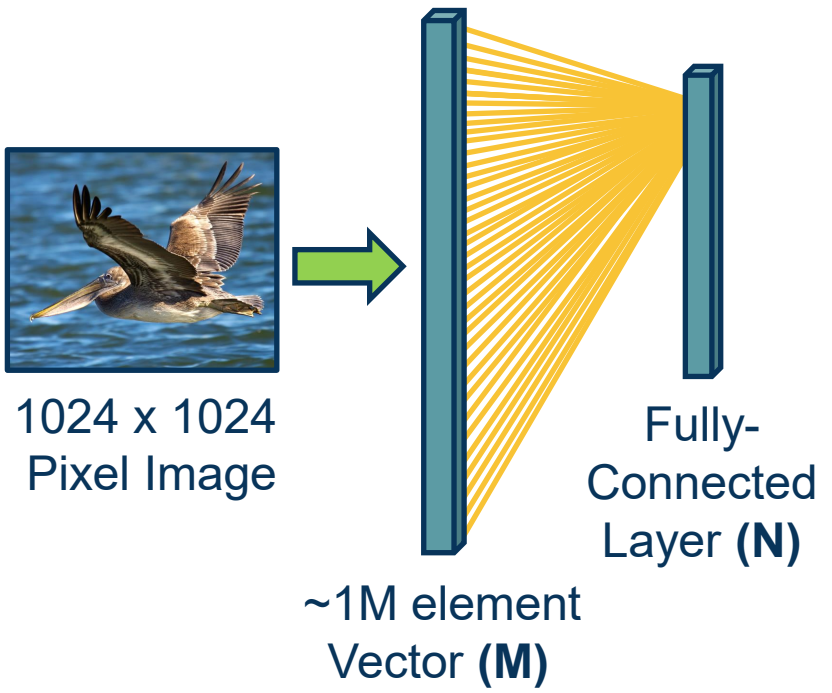
Summary

Key takeaway: **Initialization matters!**

- Determines the **activation** (output) statistics, and therefore **gradient statistics**
- If gradients are **small**, no learning will occur and no improvement is possible!
- Important to reason about **output/gradient statistics** and analyze them for new layers and architectures

Convolution & Pooling

The connectivity in linear layers **doesn't always make sense**



How many parameters?

- M*N (weights) + N (bias)

Hundreds of millions of parameters **for just one layer**

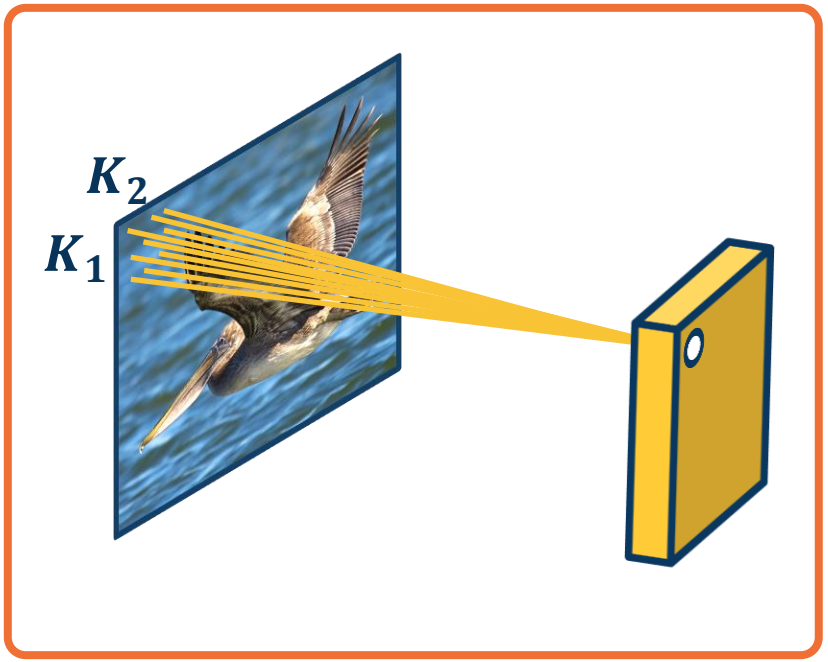
More parameters => More data needed

Is this necessary?

Image features are spatially localized!

- Smaller features repeated across the image
  - Edges
  - Color
  - Motifs (corners, etc.)
- No reason to believe one feature tends to appear in one location vs. another (stationarity)

Can we induce a *bias* in the design of a neural network layer to reflect this?



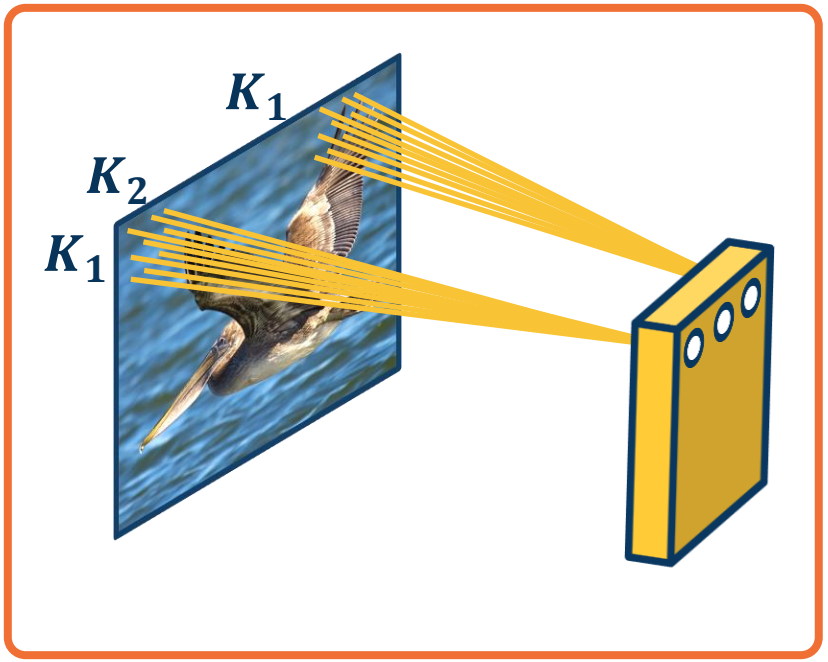
Each node only receives input from $K_1 \times K_2$ window (image patch)

- Region from which a node receives input from is called its **receptive field**

Advantages:

- Reduce parameters to $(K_1 \times K_2 + 1) * N$ where N is number of output nodes
- Explicitly maintain spatial information

Do we need to learn location-specific features?



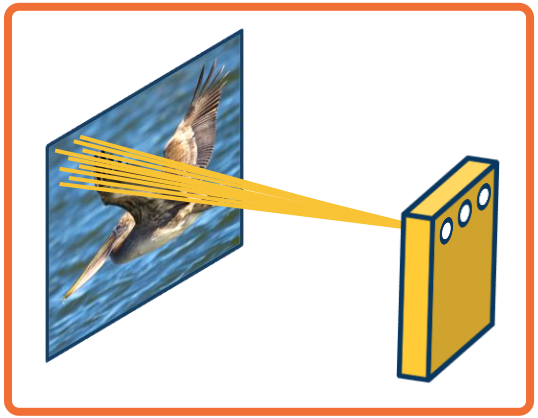
Nodes in different locations can **share** features

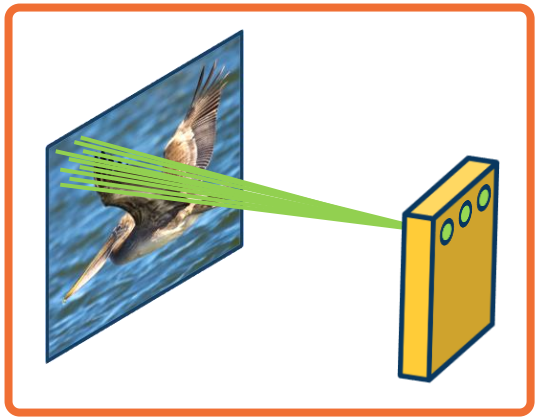
- No reason to think same feature (e.g. edge pattern) can't appear elsewhere
- Use same weights/parameters in computation graph **(shared weights)**

Advantages:

- Reduce parameters to $(\boldsymbol{K_1 \times K_2 + 1})$
- Explicitly maintain spatial information

Idea 2: Shared Weights





We can learn **many** such features for this one layer

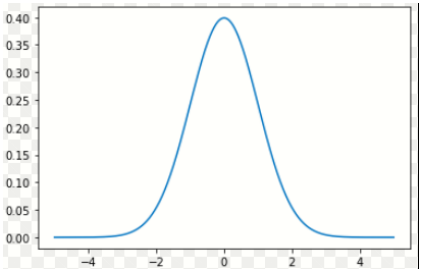
- Weights are **not** shared across different feature extractors
- **Parameters:** $(K_1 \times K_2 + 1) * M$ where M is number of features we want to learn

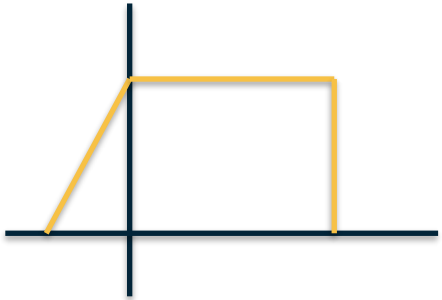
Idea 3: Learn Many Features

This operation is **extremely common** in electrical/computer engineering!

Continuous functions: $x(t)$ $w(t)$ $\rightarrow y(t)$

$$x(t) = e^{-\frac{t-t_0}{\sigma^2}}$$





$$y(t) = (x * w)(t) = \int_{-\infty}^{\infty} x(t-a)w(a)da$$

$$= (w * x)(t) = \int_{-\infty}^{\infty} x(a)w(t-a)da$$

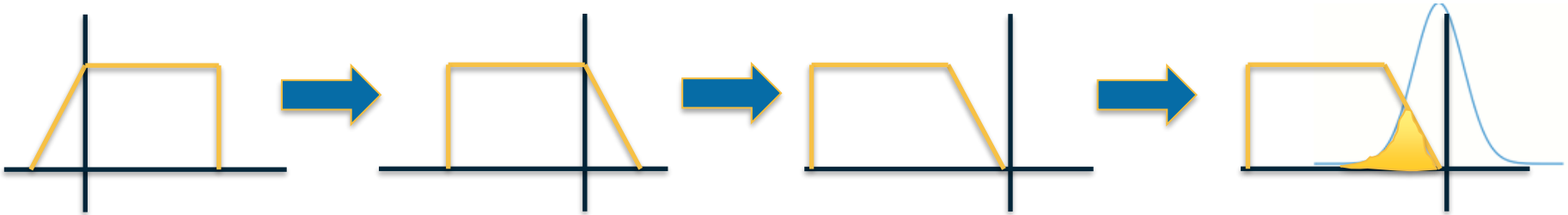
From https://en.wikipedia.org/wiki/Convolution

This operation is **extremely common** in electrical/computer engineering!

Continuous functions: $x(t)$ $w(t)$ $y(t)$

$$y(t) = (w * x)(t) = \int_{-\infty}^{\infty} x(a)w(t-a)da$$

What is $w(t-a)$?



$w(a)$

$w(-a)$

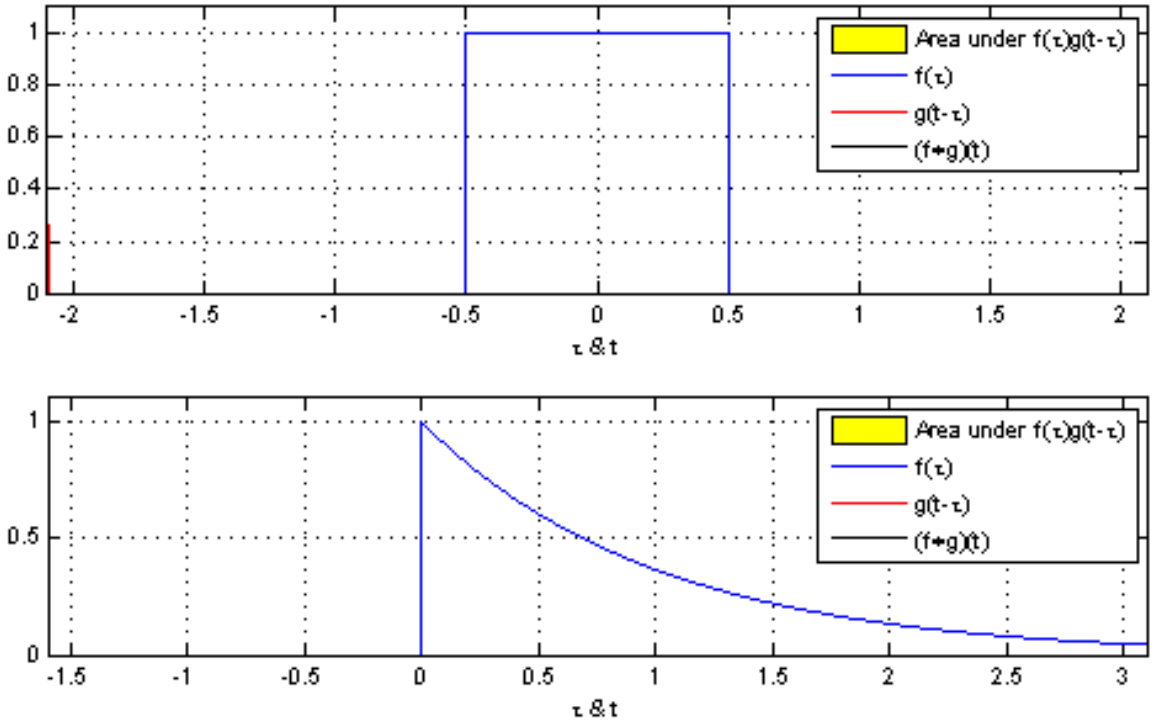
Flip

$w(-(a-t))$

Flip+Translate

$y(t)$ for one t is overlap (integral) -> y is function describing this overlap over stride

This operation is **extremely common** in electrical/computer engineering!



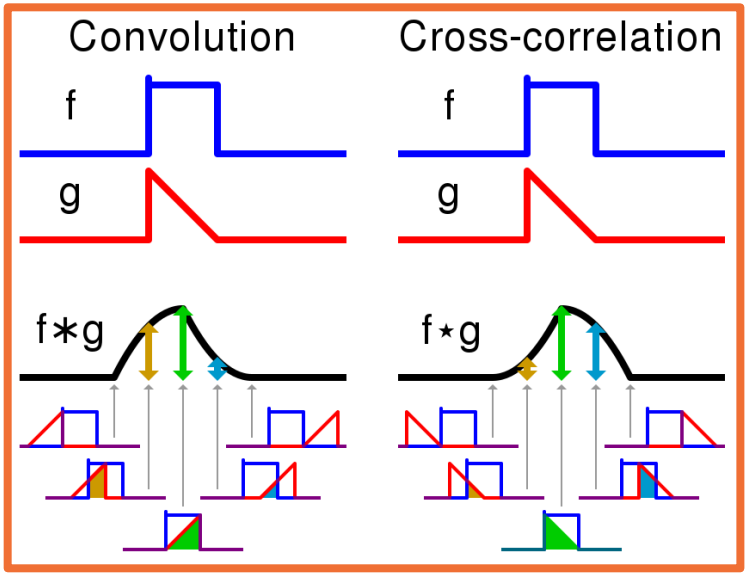
From https://en.wikipedia.org/wiki/Convolution

This operation is **extremely common** in electrical/computer engineering!

In mathematics and, in particular, functional analysis, **convolution** is a mathematical operation on two functions f and g producing a third function that is typically viewed as a modified version of one of the original functions, giving the area overlap between the two functions as a function of the amount that one of the original functions is translated.

Convolution is similar to **cross-correlation**.

It has **applications** that include probability, statistics, computer vision, image and signal processing, electrical engineering, and differential equations.



Visual comparison of **convolution** and **cross-correlation.**

From https://en.wikipedia.org/wiki/Convolution

Notation: $F \otimes (G \otimes I) = (F \otimes G) \otimes I$

1D Convolution

$$y_k = \sum_{n=0}^{N-1} h_n \cdot x_{k-n}$$

$$y_0 = h_0 \cdot x_0$$
$$y_1 = h_1 \cdot x_0 + h_0 \cdot x_1$$
$$y_2 = h_2 \cdot x_0 + h_1 \cdot x_1 + h_0 \cdot x_2$$
$$y_3 = h_3 \cdot x_0 + h_2 \cdot x_1 + h_1 \cdot x_2 + h_0 \cdot x_3$$
$$\vdots$$

2D Convolution

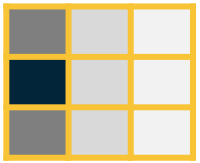
$$K = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

2D Convolution

Image

Kernel (or filter)

$$K = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$



Output / filter / feature map

We will make this convolution operation **a layer** in the neural network

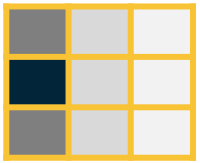
- Initialize kernel values randomly and optimize them!
- These are our parameters (plus a bias term per filter)

2D Convolution

Image

Kernel (or filter)

$$K = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$



Output / filter / feature map

- **Convolution:** Start at end of kernel and move back
- **Cross-correlation:** Start in the beginning of kernel and move forward (same as for image)

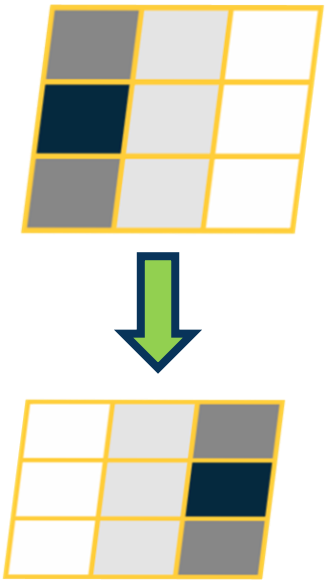
An **intuitive interpretation** of the relationship:

- Take the kernel, and rotate 180 degrees along center (sometimes referred to as “flip”)
- Perform cross-correlation
- (Just dot-product filter with image!)

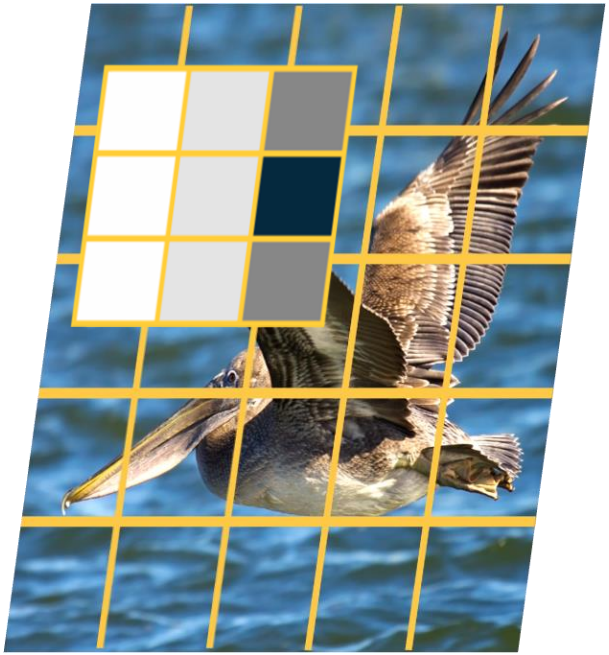
$$K = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

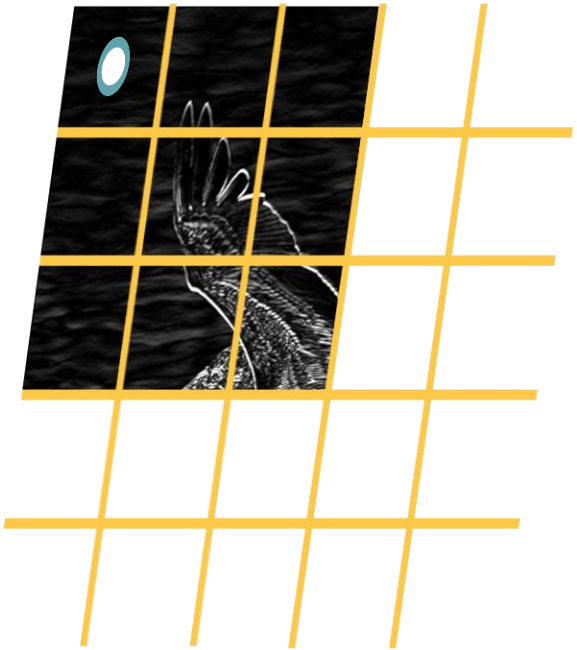
$$K' = \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

1. Flip kernel (rotate 180 degrees)

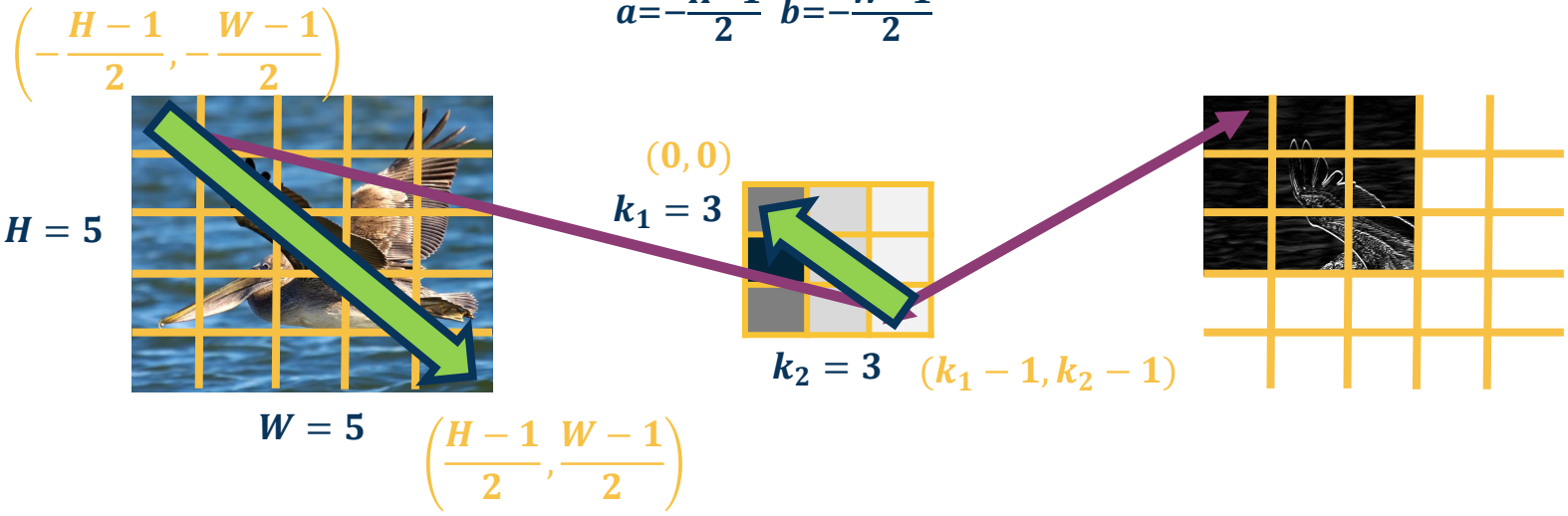


2. Stride along image





$$y(r,c) = (x * k)(r,c) = \sum_{a=-\frac{H-1}{2}}^{\frac{H-1}{2},} \sum_{b=-\frac{W-1}{2}}^{\frac{W-1}{2},} x(a,b)\, k(r-a, c-b)$$



$$y(0,0) = x(-2,-2)k(2,2) + x(-2,-1)k(2,1) + x(-2,0)k(2,0) + x(-2,1)k(2,-1) + x(-2,2)k(2,-2) + \ldots$$

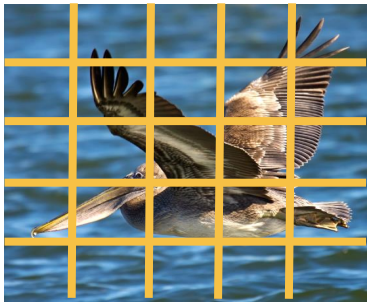
Mathematics of Discrete 2D Convolution

Georgia Tech

$$y(r,c) = (x * k)(r,c) = \sum_{a=-\frac{K_1-1}{2}}^{\frac{k_1-1}{2},} \sum_{b=-\frac{k_2-1}{2}}^{\frac{k_2-1}{2},} x(r-a, c-b)\, k(a,b)$$

$(0,0)$

$H = 5$

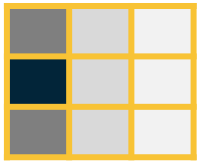


$W = 5$

$(H-1, W-1)$

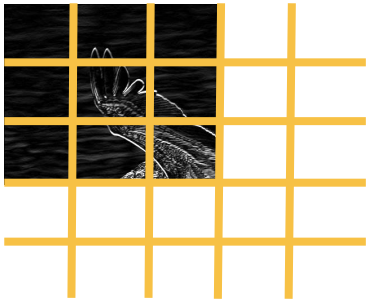
$(-\frac{k_1-1}{2}, -\frac{k_2-1}{2})$

$k_1 = 3$



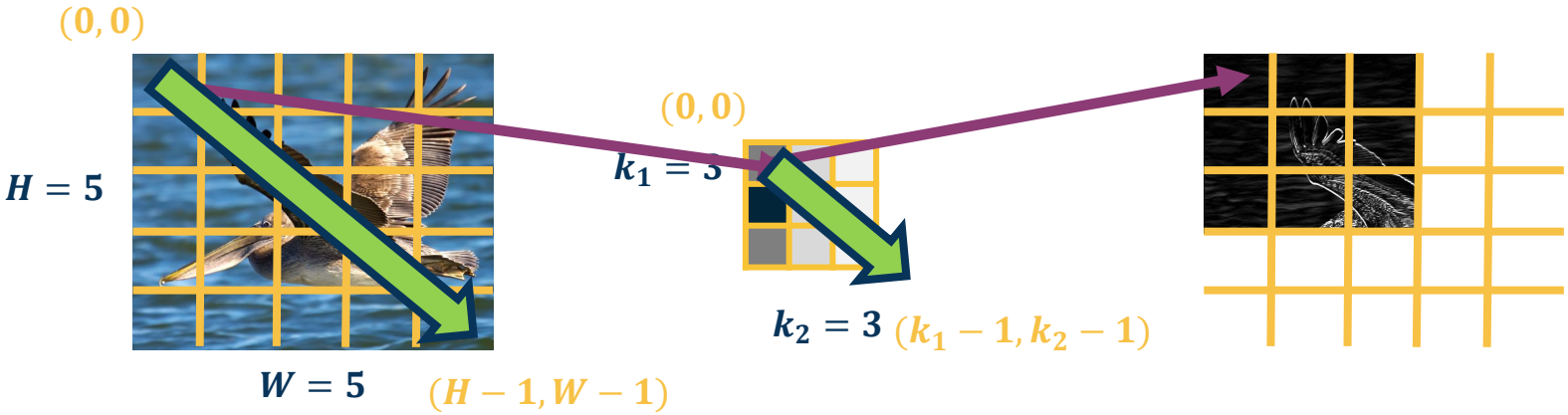
$k_2 = 3$

$(\frac{k_1-1}{2}, \frac{k_2-1}{2})$



Centering Around the Kernel

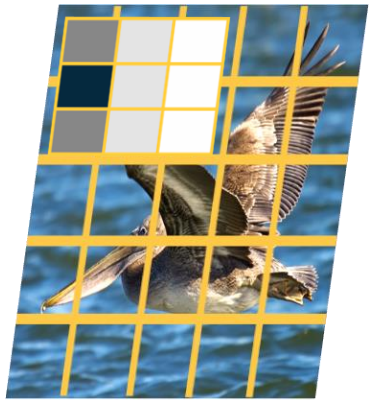
$$y(r,c) = (x * k)(r,c) = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} x(r+a, c+b)\, k(a,b)$$

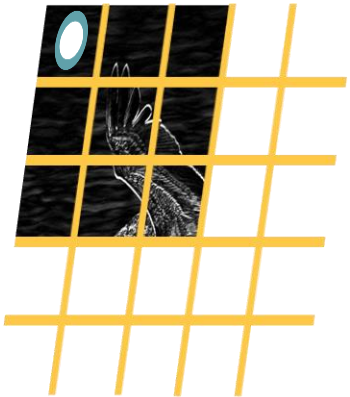


Since we will be learning these kernels, this change does not matter!

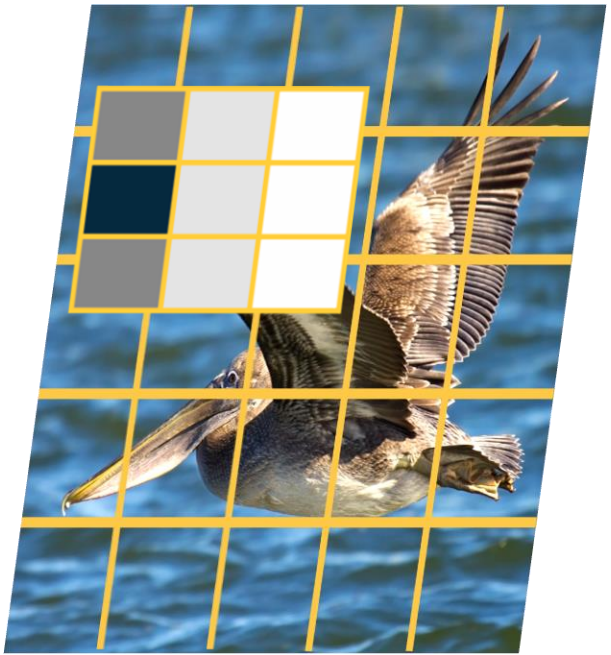
$$X(0:2,0:2) = \begin{bmatrix} 200 & 150 & 150 \\ 100 & 50 & 100 \\ 25 & 25 & 10 \end{bmatrix} \qquad K' = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \Rightarrow X(0:2,0:2) \cdot K' = 65 + \text{bias}$$

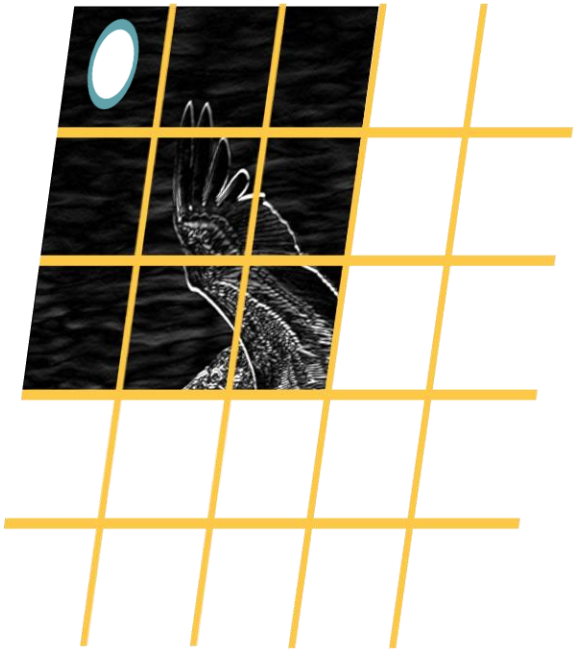
Dot product
(element-wise multiply and sum)



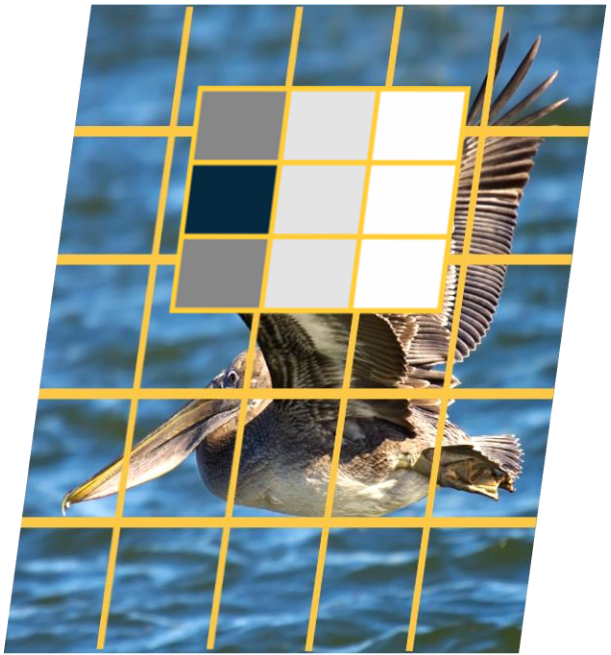


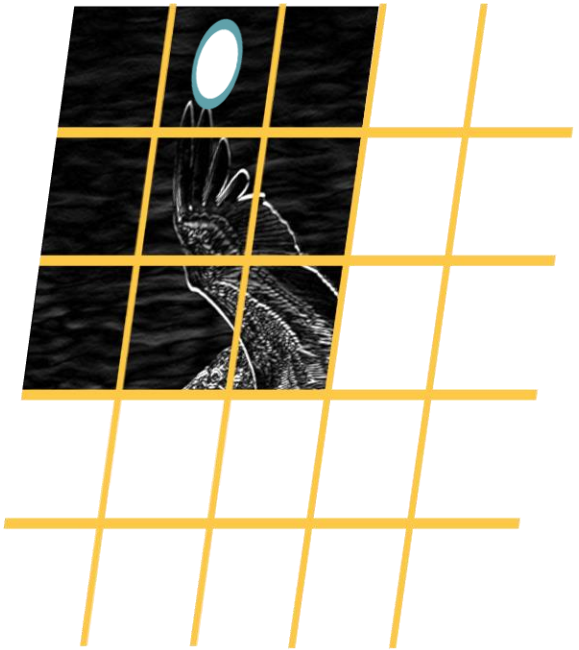
Cross-Correlation



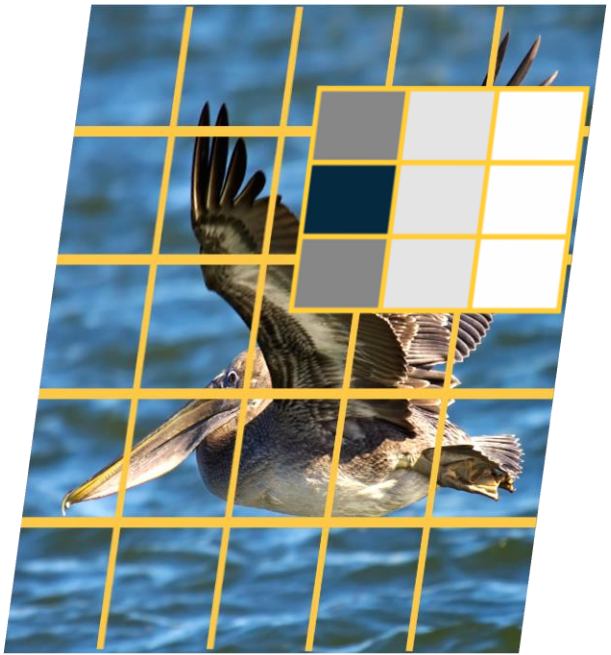


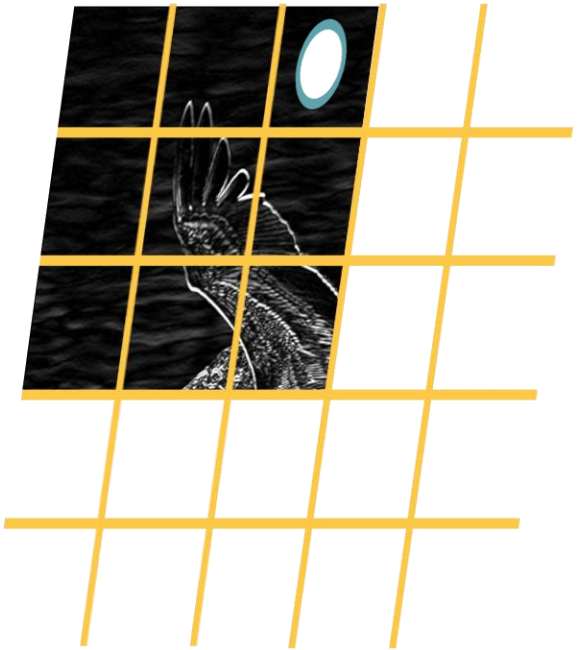
Convolution and Cross-Correlation



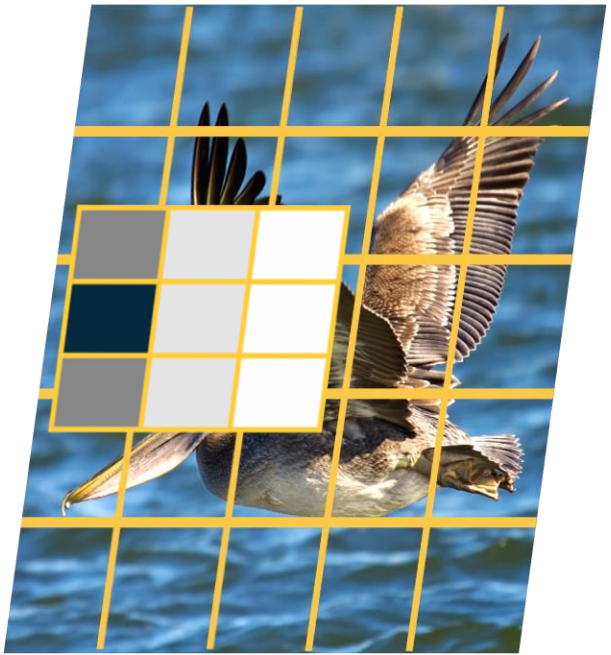


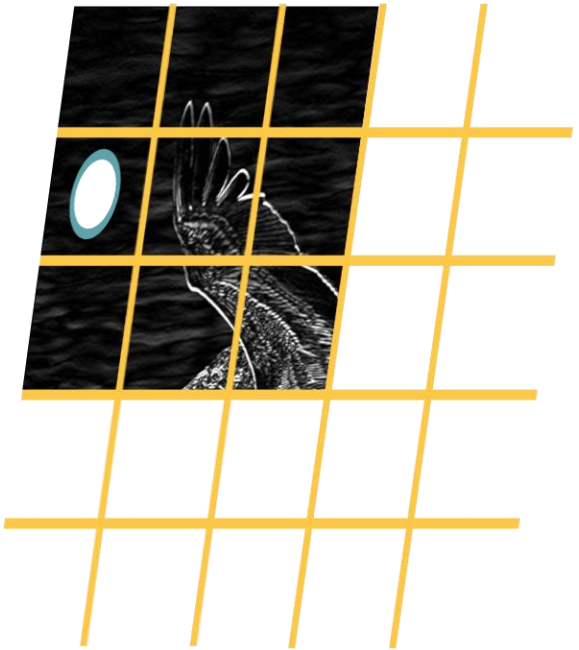
Convolution and Cross-Correlation



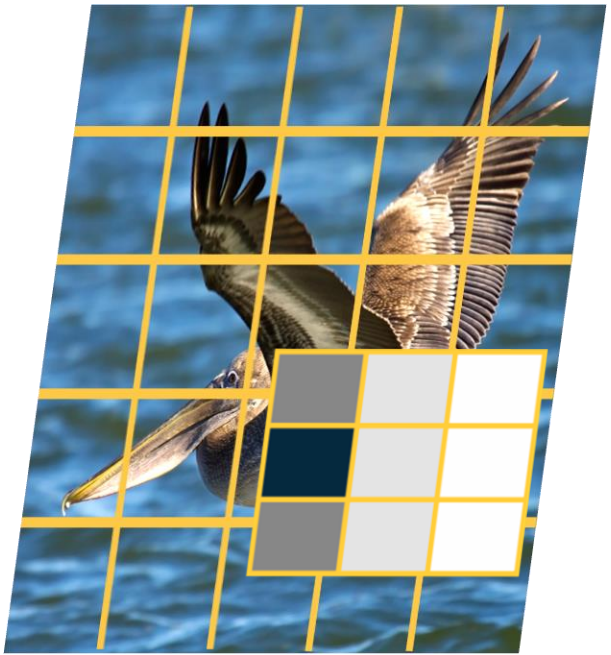


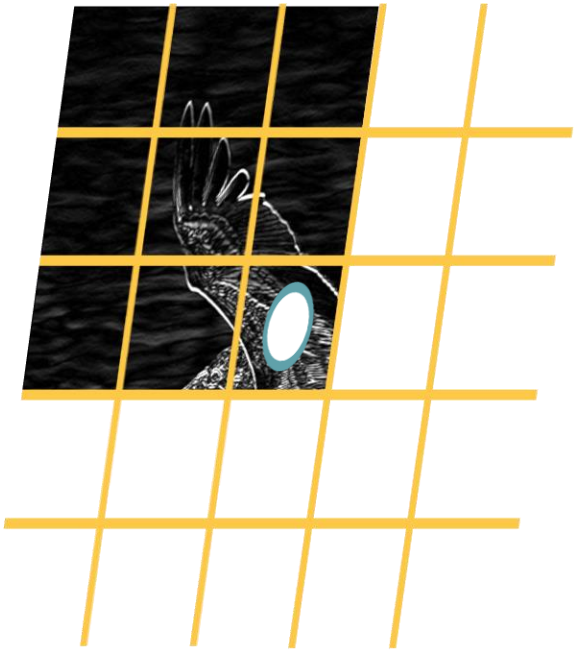
Convolution and Cross-Correlation





Convolution and Cross-Correlation





Convolution and Cross-Correlation

Why Bother with Convolutions?

Convolutions are just **simple linear operations**

Why bother with this and not just say it's a linear layer with small receptive field?

- There is a **duality** between them during backpropagation
- Convolutions have **various mathematical properties** people care about
- This is **historically** how it was inspired

Georgia Tech

Input & Output Sizes

Convolution Layer Hyper-Parameters

Parameters

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution. Default: 1
- **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input. Default: 0
- **padding_mode** (*string, optional*) – `'zeros'`, `'reflect'`, `'replicate'` or `'circular'`. Default: `'zeros'`

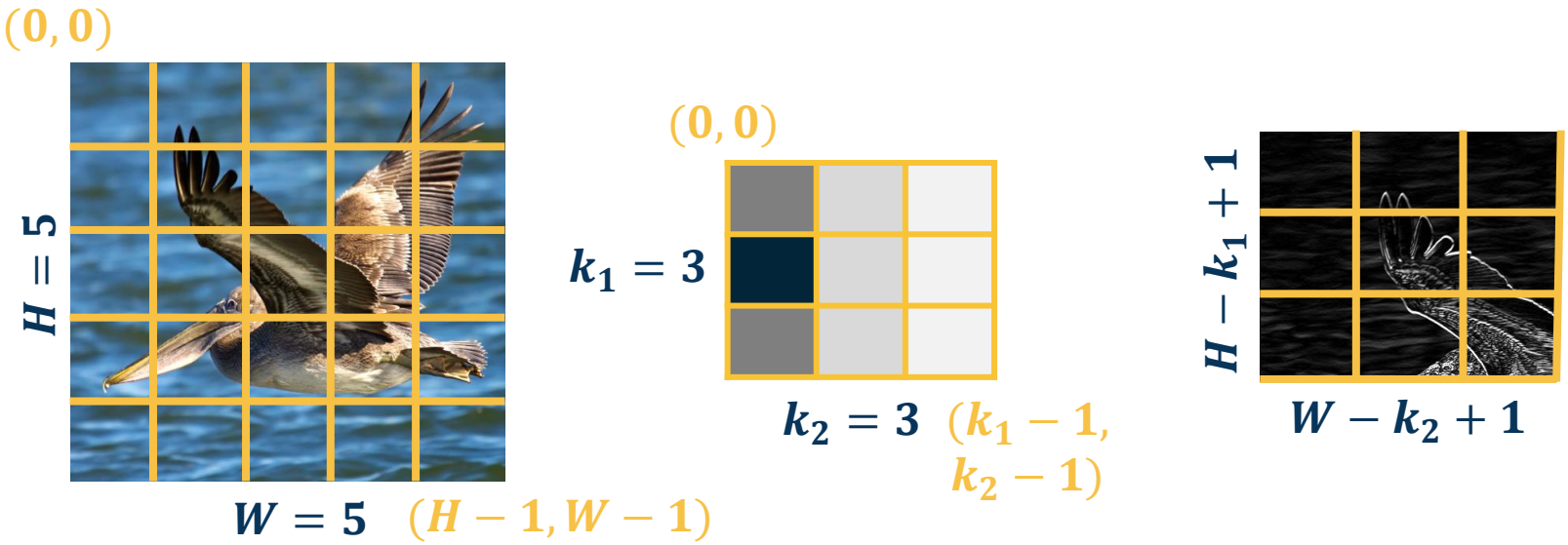
Convolution operations have several hyper-parameters

From: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html#torch.nn.Conv2d

Georgia Tech

Output size of vanilla convolution operation is $(H - k_1 + 1) \times (W - k_2 + 1)$

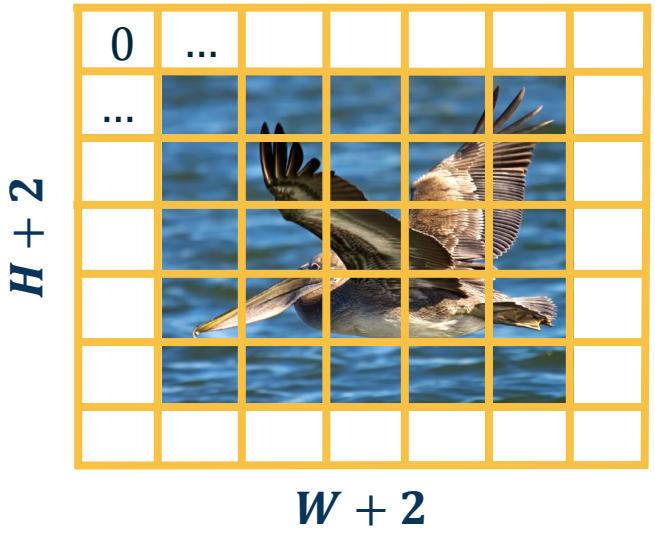
- This is called a **"valid" convolution** and only applies kernel within image

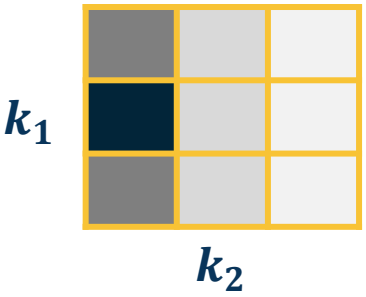


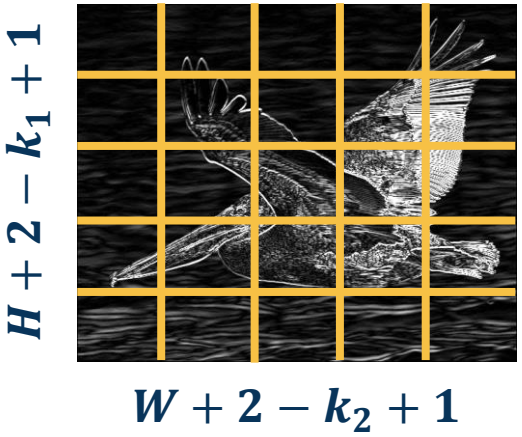
Valid Convolution

We can **pad the images** to make the output the same size:

- Zeros, mirrored image, etc.
- Note padding often refers to pixels added to **one size** ($\mathbf{P = 1}$ **here**)



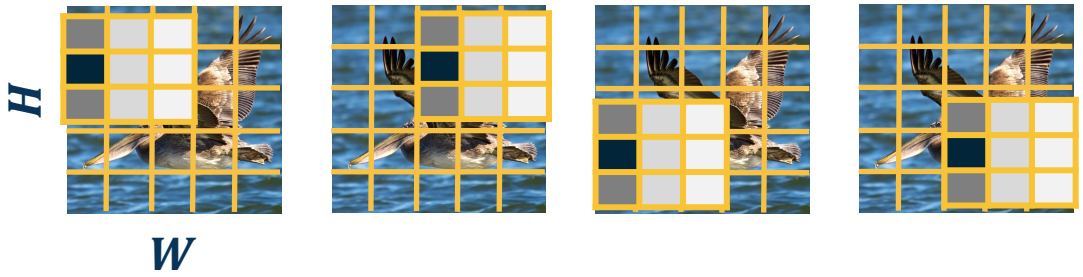


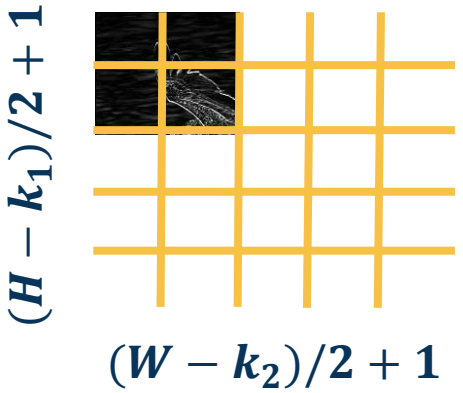


We can move the filter along the image using larger steps **(stride)**

- This can potentially result in **loss of information**
- Can be used for **dimensionality reduction** (not recommended)

Stride = 2 (every other pixel)



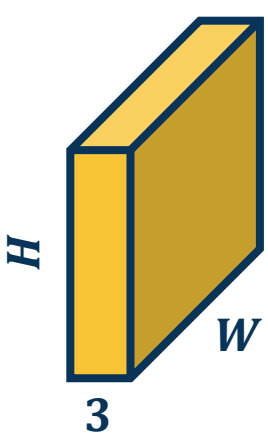


Stride can result in **skipped pixels**, e.g. stride of 3 for 5x5 input

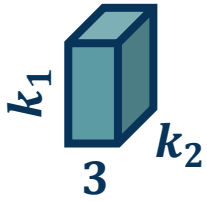
Invalid Stride

We have shown inputs as a **one-channel image** but in reality they have three channels (red, green, blue)

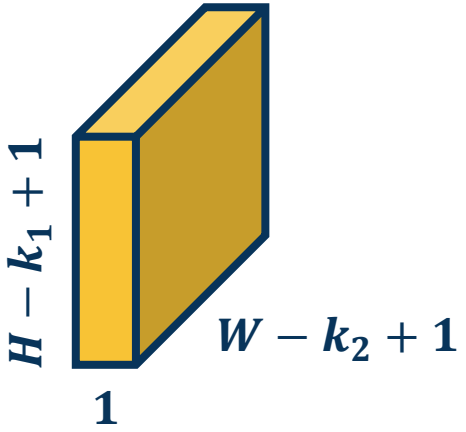
- In such cases, we have **3-channel kernels**!



Image



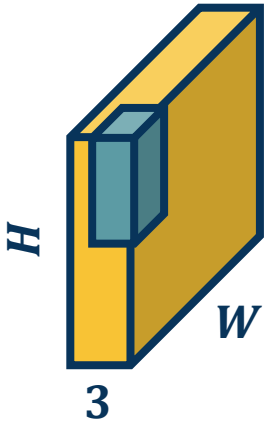
Kernel



Feature Map

We have shown inputs as a **one-channel image** but in reality they have three channels (red, green, blue)

- In such cases, we have **3-channel kernels**!



Image

Similar to before, we perform **element-wise multiplication** between kernel and image patch, summing them up **(dot product)**

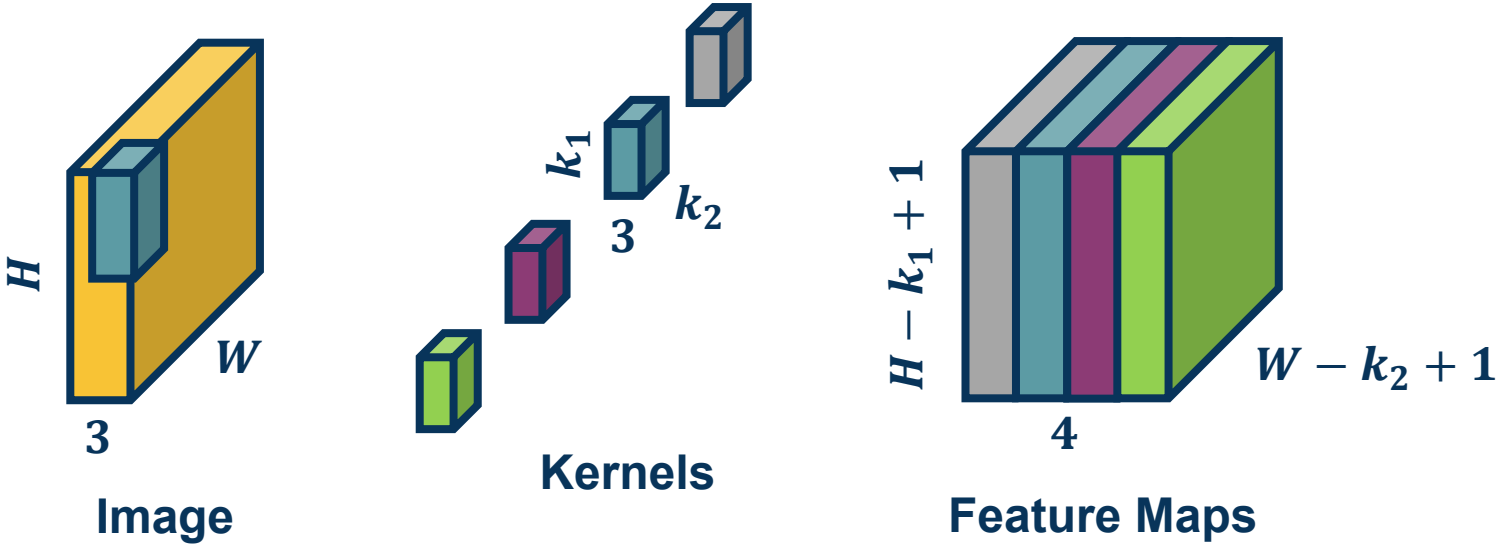
- Except with $\boldsymbol{k_1 * k_2 * 3}$ values

Operation of Multi-Channel Input

We can have **multiple kernels per layer**

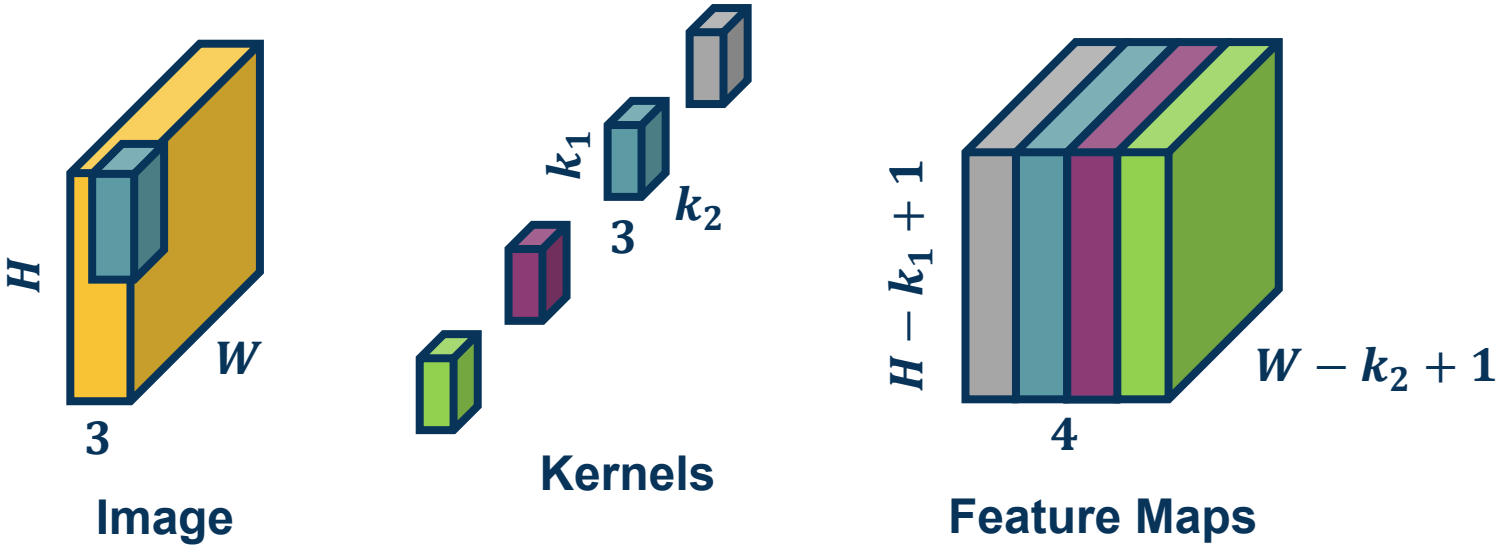
- We stack the feature maps together at the output

Number of channels in output is equal to *number of kernels*



Number of parameters with N filters is: $N * (k_1 * k_2 * 3 + 1)$

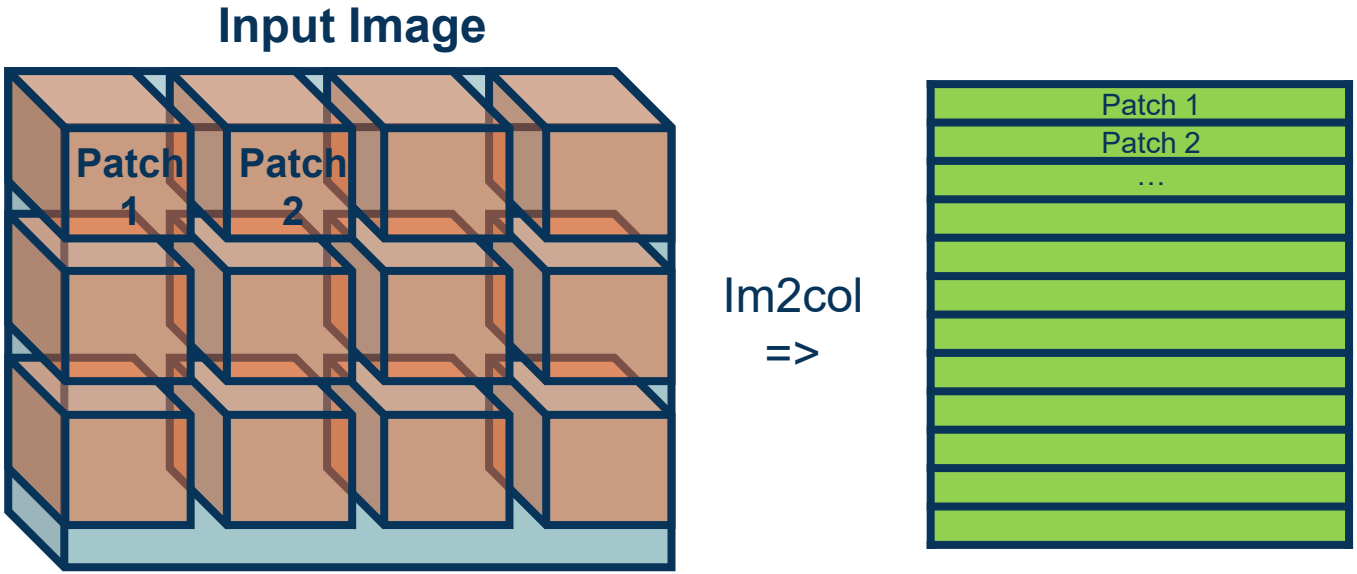
- Example:
  $k_1 = 3, k_2 = 3,\ N = 4$ *input channels* $= 3$, then $(3 * 3 * 3 + 1) * 4 = 112$



Number of Parameters

Just as before, in practice we can **vectorize** this operation

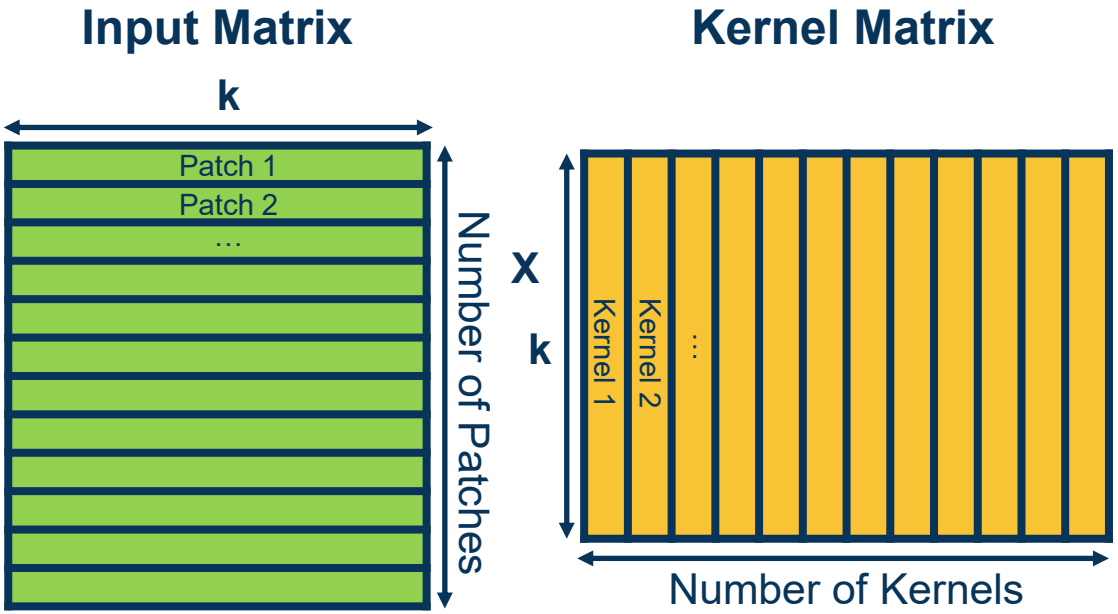
- **Step 1:** Lay out image patches in vector form (note can overlap!)



Adapted from: https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/

Just as before, in practice we can **vectorize** this operation

- **Step 2**: Multiple patches by kernels



Adapted from: https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/

- We will have a new layer: Convolution layer
  - Mathematical way of representing a strided filter
    - Equivalent view: Each output node is connected to window, not all input pixels
  - Kernels/filters/features are learned
  - Implementation is actually cross-correlation! (but it doesn't matter)

- Next time: How do we compute the gradients across this layer?
  - Need to reason about what input/weight pixel is affecting what output pixel!